ASSO - *Nginx*

Homework 6

Team 33
Guilherme de Matos Ferreira de Almeida
João Pedro Carvalho Moreira
Jorge Daniel de Almeida Sousa
Lia da Silva Linhares Vieira
Nuno Afonso Anjos Pereira

U. PORTO

Masters in Informatics and Computing Engineering

25/03/2024

Contents

1 Introduction

Nginx was born out of the necessity to support web servers struggling with concurrent connections exceeding ten thousand connections. *Nginx* stands out for its efficiency and reliability, and its innovative architecture. This paper provides a comprehensive analysis of the architectural details of *Nginx*, enhanced with clear diagrams that elucidate its structural components and interactions. Furthermore, this analysis focuses on the architectural patterns, describing the fundamental principles driving *Nginx*'s design choices and highlighting the attributes that support *Nginx*'s success, like high performance and concurrency and low memory usage.

2 Background

The idea to build *Nginx* came from the need to scale web servers beyond what the technologies present at that time allowed, namely *Apache*.

Apache had been a staple in web server technology for years and its success can be attributed to its simplicity, which matched the operational needs of corporations at the time. Eventually *Apache* grew to be one of the most extensible systems available, further increasing its acceptance by many organizations looking to augment their online presence.

However, as the Internet grew, more users began accessing web resources, imposing heavier and heavier loads on web server infrastructures. The architecture *Apache* used, which consisted of a "master" process spawning child processes/threads for each request, could not keep up with the growth in network traffic due to the CPU-intensive task of creating execution contexts, new runtime environments and allocating stack and heap memory for each request handled.

This is one of the aspects that *Nginx* tried to handle from its inception: "*From the very beginning, Nginx was meant to be a specialized tool to achieve more performance, density and economical use of server resources while enabling dynamic growth of a website, so it has followed a different model.*".[1]

3 Main Architectural style: Event-Driven

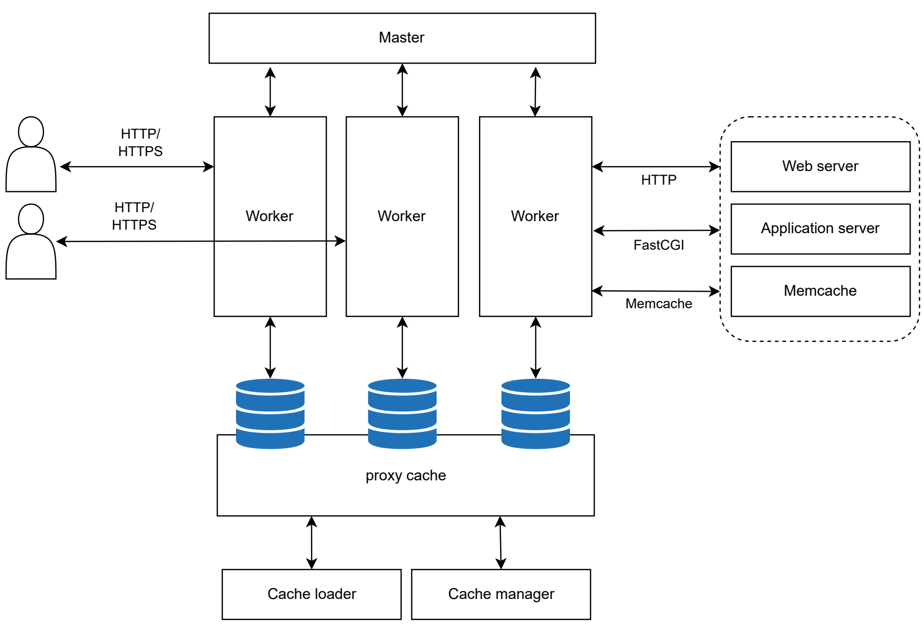
To achieve this, *Nginx* followed an **Event-Driven** approach, using a single-threaded work model.

There is one **master** process, similar to how *Apache* does it. In fact, many design choices were inspired by *Apache*'s strengths and weaknesses. However, unlike *Apache*, *Nginx*, at startup, spawns a set of processes called **workers**. These **worker** processes run a single-threaded, highly optimized run-loop which is capable of handling thousands of concurrent connections. The **master** process is responsible for spawning more **workers** if needed.

Communication between **master** and **worker** processes is handled through an event system which is built on top of shared memory principles.

Since all message-passing happens in-memory, the latency experienced with Inter-Process Communication of other kinds of applications does not even exist. Using a single-threaded work model also reduces the time spent not actually processing requests. All of these enable *Nginx* to be one of the fastest web server technologies present.

A high-level diagram of Nginx is shown below. It was adapted from the official *AOSA*[1] book and shows the high-level processes happening within *Nginx* when serving a request.



Inside each **worker** process, the run-loop relies heavily on **asynchronous task handling** and **disk and I/O optimizations** which include fully exploiting the kernel's features to minimize latency. Besides this, upon receiving a request, it is up to the kernel to distribute that request to an available **worker** process, freeing computational power from the *Nginx* processes. Also, having multiple dedicated **worker** processes allows *Nginx* to make full use of **multi-core systems**, which can parallelize the web-serving tasks more efficiently.

3.1 Pain Points of *Nginx*'s architecture

Nginx, at least as described by the AOSA[1] book, is not free of its shortcomings. One of the most prominent pain points felt is the **limited support for scripting**: while Nginx has a built-in Perl interpreter (modules can extend the Nginx core for other languages, there is already one such module for Lua) that can run custom code upon handling a request, these are not guaranteed to run to completion; in case that happens, the entire **worker** process would halt and stop processing further requests. Even worse than that, all the requests that were being handled by that **worker** would get held up. Further work has been planned and made in hopes of improving this feature. Another issue with *Nginx* is that if there is not sufficient storage performance at any given moment, a *Nginx* **worker** may block disk I/O operations. As was the case with embedded scripting, there are plans to improve this.

4 Other Patterns

Nginx's careful design comprises several other architectural patterns, which are described here. Even though these patterns are the ones that can be observed and extracted at a higher level, they are all interconnected (not necessarily coupled) so there is no clear boundary between many of them.

Besides this, some additional patterns are present but are not described here since they are not as important or useful as the ones below.

4.1 Plugin/Microkernel

Nginx's modular architecture allows developers to extend the set of web server features without modifying the *Nginx* core. The modules in *Nginx* can be considered as plugins that can be added at build time to enhance the functionality of the web server. In *Nginx*, the core serves as the minimalistic foundation responsible for essential tasks such as managing network protocols, establishing the runtime environment, and facilitating communication between various modules. This core resembles the microkernel, which provides basic services and abstracts complexities, allowing for modular expansion and customization. Conversely, the modules in *Nginx* encapsulate protocol- and application-specific functionalities, akin to microkernel extensions, enhancing the server's capabilities without compromising its core stability and performance.

4.2 Shared Repository

Nginx configuration files are stored in a central location. The main configuration file contains all of the important directives and parameters that control the server's behavior. To keep the main configuration file manageable and uncluttered, parts of the configuration can be organized into separate files. *Nginx* supports automatic inclusion of separate configuration files into the main configuration file. This implies that any changes made to the included files automatically take effect when the main configuration is loaded, simplifying configuration management and ensuring consistency across the server.

Besides this, using shared-memory principles allows the **master** process to take responsibility for managing all things related to configuration and the **worker**s can get a read-only view of this configuration object from the **master** process: effectively the **master** process serves as the "shared repository" for configuration values.

4.3 Pipes & Filters

Nginx's worker code consists of two main parts: the core and functional modules. The *Nginx* core is in charge of coordinating the data flow by keeping a tight run-loop and executing sections of modules' code at each stage of request processing. This run-loop ensures that each stage of request processing is executed efficiently. The functional modules work as pipeline filters. Every module carries out a specific task, such as reading from and writing to the network and storage, transforming content, applying filters, or passing requests to upstream servers. These modules operate sequentially, with the output of one module serving as the input to the next.

4.4 Interceptor

In *Nginx*, separate processes, and workers, handle different stages of connection processing within a highly efficient run-loop. By designating specific tasks to workers, *Nginx* employs an interception mechanism where each worker intercepts incoming connections, processes them, and manages concurrent requests.

4.5 Interpreter

The standard *Nginx* distribution supports embedding Perl scripts only. Perl is a high-level, general-purpose, interpreted, dynamic programming language. The Interpreter specifies how to evaluate Perl sentences. Modules can extend this functionality further: there is a Lua interpreter module available for Nginx.

5 Quality Attributes

Understanding the quality attributes of *Nginx* is crucial to comprehend its architecture and overall effectiveness. Some of which are:

- Configurability - *Nginx* is configured through config files. The added modules are also configured via config files.
- Scalability - Deliver tens of thousands of concurrent connections on a server.
- Modularity - Modules allow the extension of *Nginx*'s core. There are different types of modules: core modules, event modules, phase handlers, protocols, variable handlers, filters, upstream, and load balancers.
- Extensibility - New features were added (i.e. FastCGI, uswgi), use of distributed memory object caching systems (e.g. memcached) and reverse proxies. Modules can be developed to extend *Nginx*'s core functionalities.
- Modifiability - The code is open source, so anyone can modify it to adapt to their own use.
- Usability - The config files follow a C-style syntax and formatting, making it easy to use for the majority of *Nginx*'s end users.
- Maintainability - The config files can be easily automated, due to following C-style conventions.
- Portability - *Nginx* runs both on Windows and Unix systems.Although, *Nginx* for windows is more like a proof-of-concept rather than a fully functional port.

6 Conclusion

In conclusion, *Nginx*'s architecture represents a groundbreaking approach to web server design, characterized by its efficient event-driven model, master-worker paradigm, and modular structure. The incorporation of architectural patterns such as the plugin model, shared repository, and pipes & filters further enhances its flexibility and scalability. *Nginx*'s architecture excels in key quality attributes including configurability, scalability, and modularity making it a preferred choice for handling modern web traffic demands. This work has provided us with a deeper understanding of the *Nginx* architecture and the patterns that have contributed to its success.

References

[1] Andrew Alexeev. nginx. *Architecture of Open Source Applications 2nd volume*, 2012.